Estimate, as a function of the grid size n, the amount of space used by PercolationVisualizer (PROGRAM 2.4.3) with the vertical percolation detection (PROGRAM 2.4.2). Extra credit: Answer the same question for the case where the recursive percolation detection method (PROGRAM 2.4.5) is used.

Code and Space Analysis

```
public class PercolationVisualizer {
    public static void main(String[] args) {
        int n      = Integer.parseInt(args[0]);
        double p   = Double.parseDouble(args[1]);
        int trials = Integer.parseInt(args[2]);

        // repeatedly created n-by-n matrices and display them
using standard draw
        StdDraw.enableDoubleBuffering();
        for (int t = 0; t < trials; t++) {
            boolean[][] open = Percolation.random(n, p);
            StdDraw.clear();
            StdDraw.setPenColor(StdDraw.BLACK);
            Percolation.show(open, false);
            StdDraw.setPenColor(StdDraw.GRAY);
            boolean[][] full = Percolation.flow(open);
            Percolation.show(full, true);
            StdDraw.show();
            StdDraw.pause(1000);
        }
    }
}
```

To estimate the amount of space used by this program, we need to consider the memory consumed by the data structures and variables used within the `main` method of the `PercolationVisualizer` class.

Here's a breakdown of the space usage:

1. **Variables:**

   - `n`, `p`, `trials` (int and double): These primitive types use a fixed amount of space.
     - `int` variables (`n` and `trials`): 4 bytes each
     - `double` variable (`p`): 8 bytes
2. **Boolean Arrays:**

   - The `open` and `full` arrays are `n x n` boolean arrays.
     - A boolean in Java typically uses 1 byte of space.
     - Two `n x n` boolean arrays would use `2 * n * n` bytes of space.
3. **Other Overheads:**

- There might be some additional overhead for the array objects themselves, including object headers and any possible padding. The exact amount can depend on the JVM and platform, but a rough estimate for the overhead per array object is about 16 bytes.

Let's compute the total space usage step by step:

Variables

- 2 `int` variables: $2 \times 4 = 8$ bytes
- 1 `double` variable: 8 bytes

Total for variables: $8 + 8 = 16$ bytes

Boolean Arrays

- Size of one `n x n` boolean array: $n \times n$ bytes
- Total size for two `n x n` boolean arrays: $2 \times n \times n$ bytes
- Overhead per array: 16 bytes
- Total overhead for two arrays: $2 \times 16 = 32$ bytes

Total for boolean arrays: $2 \times n \times n + 32$ bytes

Total Space Estimate

Combining the variables and the boolean arrays:

Total space used:
$16 \text{ bytes (variables)} + 2 \times n \times n \text{ bytes (arrays)} + 32 \text{ bytes (overhead)}$

Simplifying: $16 + 2n^2 + 32 = 2n^2 + 48$

Thus, the estimated amount of space used by this program, as a function of the grid size n, is approximately:

$2n^2 + 48 \text{ bytes}$

For large values of n, the 48 bytes of overhead will be negligible, and the space usage will be dominated by the $2n^2$ term.

First Method (Recursive `flow` method)

```
// determine set of full sites using depth first search
public static void flow(boolean[][] isOpen, boolean[][] isFull, int
i, int j) {
    int n = isOpen.length;

    // base cases
```

```
    if (i < 0 || i >= n) return;    // invalid row
    if (j < 0 || j >= n) return;    // invalid column
    if (!isOpen[i][j]) return;      // not an open site
    if (isFull[i][j]) return;       // already marked as full

    // mark i-j as full
    isFull[i][j] = true;

    flow(isOpen, isFull, i+1, j);   // down
    flow(isOpen, isFull, i, j+1);   // right
    flow(isOpen, isFull, i, j-1);   // left
    flow(isOpen, isFull, i-1, j);   // up
}
```

Space Usage:

- Variables: 16 bytes
- Boolean Arrays: $2n^2$ bytes
- Array Overhead: 32 bytes (2 arrays)
- Recursive Call Stack: $32n$ bytes

Total Space Estimate: $2n^2 + 32n + 48 \text{ bytes}$

Second Method (Wrapper calling Recursive `flow` method)

```
public static boolean[][] flow(boolean[][] isOpen) {
    int n = isOpen.length;
    boolean[][] isFull = new boolean[n][n];
    for (int j = 0; j < n; j++) {
        flow(isOpen, isFull, 0, j);
    }
    return isFull;
}
```

Space Usage:

- Variables: 16 bytes
- Boolean Arrays: $3n^2$ bytes
- Array Overhead: 48 bytes (3 arrays)
- Recursive Call Stack: $32n$ bytes

Total Space Estimate: $3n^2 + 32n + 64 \text{ bytes}$

Third Method (Iterative `flow` method)

```
public static boolean[][] flow(boolean[][] isOpen) {
    int n = isOpen.length;
    boolean[][] isFull = new boolean[n][n];
    for (int j = 0; j < n; j++)
        isFull[0][j] = isOpen[0][j];
    for (int i = 1; i < n; i++)
        for (int j = 0; j < n; j++)
```

```
            isFull[i][j] = isOpen[i][j] && isFull[i-1][j];
    return isFull;
}
```

Space Usage:

- Variables: 16 bytes
- Boolean Arrays: $3n^2$ bytes
- Array Overhead: 48 bytes (3 arrays)
- No Recursive Call Stack (iterative)

Total Space Estimate: $3n^2 + 64 \text{ bytes}$

Side-by-Side Comparison

Feature	First Method (Recursive)	Second Method (Wrapper + Recursive)	Third Method (Iterative)
Code	See above	See above	See above
Variables	16 bytes	16 bytes	16 bytes
Boolean Arrays	$2n^2$ bytes	$3n^2$ bytes	$3n^2$ bytes
Array Overhead	32 bytes	48 bytes	48 bytes
Recursive Call Stack	$32n$ bytes	$32n$ bytes	0 bytes
Total Space Estimate	$2n^2 + 32n + 48$ bytes	$3n^2 + 32n + 64$ bytes	$3n^2 + 64$ bytes

Conclusion

- **First Method (Recursive):** The most space-efficient for smaller values of n, but can grow with the call stack for larger n.
- **Second Method (Wrapper + Recursive):** Uses more space due to the additional `isFull` array and the recursive call stack.
- **Third Method (Iterative):** Avoids the recursive call stack, making it more space-efficient than the second method but still less efficient than the first for smaller n. For large n, it becomes competitive with the first method.

For very large grid sizes, the space usage of the third method is preferable as it avoids the overhead associated with recursion.